**Kapcsolatok**

**(kapcsolat def: entitásnak entitás avagy entitás kollekció atrija van; kategorizálás: számosság és irányítottság: 1.1, 1-több, több-több és egyirányú/kétirányú=oda-vissza hivatkozás)**

*Hogyan lehet az entitások között kapcsolatokat definiálni?*

**Kapcsolat: az entitásnak egy attribútuma másik entitás típusú, vagy másik entitásnak valamilyen kollekciója**

**Milyen tulajdonság alapján lehet a kapcsolatokat kategorizálni?**

*Számosság alapján*

* 1-1; például: alkalmazott és a hozzá tartozó parkolóhely
* 1-több; például: alkalmazott és telefonszámai
* több-több; például: alkalmazott és projekt

Javas reprezentációk:

1-1: egyszerű attribútumot veszünk föl

1-több: kollekció attribútumot veszünk föl

több-több: kollekció attribútumot veszünk föl mindkét oldalon

*Irányítottság alapján*

a kapcsolatban részt vevő két entitás közül melyik hivatkozik melyikre? (melyiknek attribútuma a másik)

* egyirányú: csak egyik hivatkozik a másikra attribútumként
* kétirányú: oda vissza hivatkozik mindkettő a másikra

**Gyakrabban használt kombinációk**

**(egyirányú 1-1: @OneToOne mindkét oldara;, kétirányú 1-1: @OneToOne mindkét oldalra és mappedby atri az egyik oldalra; kétirányú 1-több: @OneToMany és @ManyToOne + mappedby; metódusok az irányok beállítására; csak proxy betöltés getReferencel, hogy az id-t külső kulcsként fel tudja használni a másik entitás lementéséhez)**

a kapcsolatot reprezentáló nem egyszerű, hanem kapcsolati attribútumokat annotációkkal látjuk el; ezekkel lesznek beállítva a kulcsok (külső kulcs a másik entitás táblájára) és ha más oszlopnevet akarunk a kulcsnak (@JoinColumn)

**Egyirányú 1-1 kapcsolat**

pl. Employee és hozzá tartozó ParkingPlace

**@OneToOne** annotáció

**külső kulcs nevének személyre szabása** esetén **@JoinColumn** annotáció

**Kétirányú 1-1 kapcsolat**

mindkét oldal hivatkozik a másikra

mindkét atrira **@OneToOne** annotáció

**külső kulcs nevének személyre szabása** esetén **@JoinColumn** annotáció

**mappedBy attribútum az egyik oldalon a @OneToOne annotációban** azért, hogy a JPA tudja, hogy ez ugyanannak a kapcsolatnak a két oldala, nem pedig két külön kapcsolat**; tartalmaznia kell a másik oldalon szereplő, visszamutató atrinak a nevét;** ez definiálja azt is, hogy a külső kulcs melyik táblában fog elhelyezkedni

**külön metódus a két irány beállítására** (egyik oldalon is pl egy add meg a másikon is)

**Kétirányú 1-több kapcsolat**

pl. Employee és hozzá tartozó telefonszámok

Az egyke entitásban felvett kollekció atrira **@OneToMany** annotáció

a másik oldalon az atrira **@ManyToOne** annotáció

**mappedBy attribútum megadása a @OneToMany annotációnak,** hogy a JPA össze tudja kötni, hogy ez egy kapcsolatnak a két iránya, a másik entitásban lévő attribútum nevével (=mappedby = ”Employee”)

**metódus implementálása, ami felveszi az az entitás kollekció atrijába az objektumokat és beállítja a másik oldalon is a kapcsolatot** (phone numbereket felvesz a setbe az employee oldalon és a phone numberekhez beállítja az employeet)

**getReference**

Entity managernek van egy segédmetódusa, a .getReference(), amivel **nem kell betölteni az egész entitást, ez csak egy proxit tölt be (=**egy olyan entitást, ami úgy viselkedik, mint az eredeti, de nem nyúl adatbázishoz, **csak az idját szedi ki, amit külső kulcsként felhasználva menti majd le az adatbázisba a másik oldali kapcsolatot**, a telefonszámot az employee id-val

az adott **proxy objektumot úgy kapjuk meg,** hogy **meghívjuk a getReferencet**, **paraméternek** átadjuk a **classt** és az **azonosítót**



**Cascade, Orphan removal, Sorrendezés, Szülő oldal, Lazy, Entity Graph**

**(cascade def: a műveleteket végezze el a kapcsolódó entitásokon is, @xxToxx annotáció atrijaként átadva (cascade=xx); árván maradt gyerek törlése (orphan removal) automatikusan: @OneToMany(OrphanRemoval = true); Sorrendezés lista atri esetén: @OrderedBy+atri vagy külön mező/atri a rendezésre @OrderColumn-al; owner side-> joincolumn, inverse side-mappedby; default betöltései az @ElementCollecitonoknek: 1-1:eager, több-1:eager, 1-több:Lazy, több-több:lazy**

**entity graph = milyen kapcsoló entitások kerüljenek betöltésre?; megadás hintekkel(metóduson belül?) vagy annotációkkal: @NamedEntityGraph, attributenotes, subgraphs; használat: gráf lekérdezése name alapján mapbe, a mapben lévő hintek átadása a find() 3. paramétereként)**

**Kaszkádolt műveletek**

cascade művelet jelentése: **művelet elvégzése az entitáson és a kapcsolódó entitás(ok)on is**

pl. ha az adott employeet ha elmentjük, a hozzá tartozó addresseket is mentse el

meg lehet mondani neki azt is, hogy pl. törlést is végezzen a persist mellett



**Orphan removal**

amikor egy gyermek rekord csak egy szülőhöz tartozhat, lehet egy olyan feltételünk, hogyha a szülőt eltávolítjuk, és az adott gyerek rekord az árva marad (nincs más kapcsolódása) akkor ezt törlődjön automatikusan: a @OneToManynek átadjuk az orphanRemoval = True atrit



**Sorrendezés**

**ha nem setet, hanem listát deklarálunk,** fontos lesz a sorrend; ezt adatbázisban valahogy jelölni kell; erre módok:

**@OrderBy annotáció** és **paraméterként** megadjuk, hogy **melyik attribútum szerint rendezze** be az entitásokat

**másik megoldás, hogy külön mezőt veszünk fel** a rendezendő entitásban, ami majd tárolja az adott sorrendet; **az atrira rátesszük az @OrderColumn annotácót és a JPAra bízzuk a rendezést**

**Szülő oldal**

vagy angolul owner, **owner side** owner of relationship

másik oldal: **inverse side**

az lesz az **inverse oldal, ahol a mappedBy** van

ahol a **join column** van, az pedig az **owner side**

**Lazy kapcsolat**

JPA teljesítménnyel kapcsolatos funkciója a lazy betöltés: amikor egy entitást betöltünk, nem feltétlen akarjuk, hogy a hozzá kapcsolódó összes entitás is betöltésre kerüljön, főleg bonyolult objektumstruktúra esetén; megadhatjuk, hogy az entitás lekérdezéskor milyen más entitások kerüljenek betöltésre

Kétfajta betöltés: lazy és eager

Eager – amikor betölti a kapcsolódó entitás(okat)

@ElementCollection esetén is van ilyen, alapesetben eager

a default érték függ a kapcsolatok számosságától:

1-1: default eager, több – 1: eager; 1 – több: lazy:; több-több: lazy a fetch type(a betöltés típusa)

a betöltési módot személyre tudjuk szabni a fetch attribútummal, FetchType.EAGER vagy FetchType.LAZY megadásával; nem javasolt! mert globálisan definiáljuk ezzel

**N+1 probléma**

felmerülhet olyan selectnél, amikor a lekérdezés egy entitásra, majd entitásonként a kapcsolódó entitásra irányul; túl sok select kerül kiadásra; ehelyett fetch joinos lekérdezés, ami betölti egyből a kapcsolódó entitásokat; egyetlen egy darab select lekérdezés lesz, joinnal

**LazyInitializationExcepton**

Hibernate esetén entitás betöltése után már lezárult a persistence context de még utána akarunk hozzáférni pl az employee telefonszámaihoz; nem fog menni mert már az entitás detached, a telszámok helyén meg egy proxy szerepel, ami az adatbázisból ténylegesen le akar kérdezni, de már nem tud; kiküszöbölés úgy, hogy az entitás betöltésekor töltse be a telszámokat is; selectbe kell megadni, join fetch-el

**Entity graph**

**le lehet vele írni, hogy milyen kapcsolódó entitások kerüljenek betöltésre**

megadhatjuk a find()nak és a createquery()nek is

*a query/lekérdezésnek megadhatjuk hinttel, kétféleképpen:*

* javax.persistence.fetchgraph (ne vegye figyelembe az alapértelmezett betöltési utasításokat, az annotációkat)
* javax.persistence.loadgraph (az alapértelmezettet még kiegészíti az általunk megadottakkal)

*statikus deklarációja annotációkkal történik, vagy programozottan állítjuk össze.*

**Entity graph deklarálása annotációkkal**

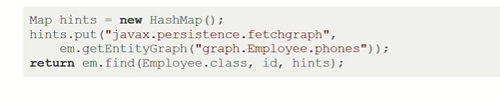
Az Employee-ra rátesszük a **@NamedEntityGraph** annotációt

megmondjuk, hogy a phones attribútumnak szeretnénk még betölteni a típusait is, és a gráfot és subgráfot nevekkel látjuk el

****

**Entity graph használata**

ezután erre egy lekérdezésben a következőképpen tudunk hivatkozni; **létrehozunk egy Mapet és lekérdezzük az előbb annotációkkal deklarált entity gráfot name alapján**, a find()nak átadjuk ezeket a hinteket harmadik paraméterként



**Kapcsolatok - gyakorlat**

1. *új osztály: ParkingPlace* -> int number; get/set @Entity és private Long id @Id, @generatedValue .IDENTITY, konstruktorok
2. *új osztály: ParkingPlaceDao*; privat entityManager

*void saveParkingPlace();* entitymanager, begin, persist(parkingplace), commit, close;

1. *ParkingPlace findParkingPLaceNumber(int number);* em, createquery(select p from..); setparameter();getsingleResult; close; return parkingplace
2. *ParkingPlaceDaoTest osztály*
3. private ParkingPlaceDao és init() -> entitymanagerFacotry példányosítás és dao atriinicializálás
4. *void testSave();* save(new ParkingPllace); lekérdezés findParkingPlaccel és assert
5. *1-1 kapcsolat létrehozása;* Employeeba Parkingplace atri és getset; @OneToOne rája
6. *testSaveEmployeeWithParkingPLace();* saveParkingPlace(new Parkingplace); new Employee(); employee.setParkingPLace(); EmployeeDao is kell a tesztbe, ezért osztályatri felvesz plusz initnél inicializál; majd saveemployee(); futtatás; **lérejött egy parkingplace\_id külső kulcs a szülő táblában**
7. test metódusba írtunk tovább assertet a getNumberrel; left outer joinnal kérdezte le a háttérben; default eager betöltés és lekérdezte
8. *1-több kapcsolat létrehozása;* mapes részt kikommentelte; új osztály: PhoneNumber; String type és String number; getset és konst; @Id Long id ,@GeneratedValue(..IDENTITY); @Table(name=.phone\_numbers)

Employeeba ezen új atri felvétele Setként @OneToMany annotációval; getset

*új test EmployeeDaoTestbe*: eredti testPhonenumbers kommentbe; új uezzel a névvel; új Phonenumberek:phonehome és work; new Employee; setPhonenumbers(=set.of..; majd saveEmloyee; **kivételt kaptunk -> olyan entitást akarunk lementeni, amiben szerepelnek olyan entitások, amelyek nem kerültek perzisztálásra;** megoldás:

Phonenumbers atrinál a @OneToManyhez megadjuk a cascade=CascaType.PERSIST;) )

**egyirányú onetomany kapcsolat; a hibernate egy kapcsolótáblát hozott létre! (employee\_id és phonenumber\_id); nekünk ez így nem ok, inkább legyen a phonenumbersben egy külső kulcs, ami az employeesra mutat:**

**ehhez kétirányú 1-több kapcsolat kell:**

1. *Phonenumber osztályba kell egy Employee atri* @ManyToOne annotációval; getset; a másik irányú kapcsolatot is be kell állítani metódussal: Employeeba új metódus: **addPhoneNumber() a kapcsolat beállítására**:
2. public void addPhoneNumber(PhoneNumber phoneNumber) {  
    if (phoneNumbers == null) {  
    phoneNumbers = new HashSet<>();  
    }  
    phoneNumbers.add(phoneNumber);  
    phoneNumber.setEmployee(this);  
   }

**test metódushoz még hozzáírás:** addphonenumber(2x); futtatás: a hibernate még az hiszi, hogy két külön kapcsolat ez; ezért ezt is definiálni kell: a onetomany oldalon a **mappedby atrival** megmondom hogy a másik entitásban hogy hívják a visszamutató atrit; futtatás: az adatbázisban csak employees és phonenumbers tálba van, és a phonenumbersben 4 oszlop van, **megjelent az employee\_id oszlop, mint külső kulcs (és már nincs kapcsolótábla, mert tudja, hogy uaz a kapcsolat a mappedby miatt)**

1. *Phonenumber atrira @JoinColumn(name=emp\_id)*
2. *Új telefonszám hozzáadása már perzisztált alkalmazotthoz*

*új metódus: void addPhoneNumber(long id, phonenumber)* EmployeeDaoba; új em, begin, employee =find(e.class, id) ; phonenumber(setEmployee(emp) és csak utána persistálás! .persist; commit és close

új test: testAddPhonenumber(); új emp, saveemp, addphonenumber, futtatás: létrejött az employee és a phonenumbersbe is az új rekord, a megfelelő külső kulccsal; a kapcsolat **owner sideja a phonenumbber, inverse az employee**

1. feleslegesen töltjük be a teljes employeet az addphonenumber metódusban, csak az idjára van szükségünk: *használjuk csak proxyt getReference()-et find helyett;* futtatáskor látjuk a háttében, hogy nem fut le a select
2. utóbbi két testre assert a visszoalvasásra: employeeanother =..findempbyid; futtatásra exception; lazy betöltéssel próbálkozik, de a lekérdezéskor már az employee detached, a pc le van zárva, a proxy már nem tud az adatbázishoz hozzányúlni; megoldás:

*új metódus az EmployeeDaoba: Employee findEmployeeByIdWithPhoneNumbers(id);* em, createquery(selct e from Employee e **join fetch e.phonenumbers where e.id = : id”, e.class**).setparameter().singleres(), close, return employee; végül assertek a phonenumbers set méretére

1. *legyen* ***sorrendiség*** *a telefonszámok között;* 2 lehetőség: employee lekérdezésekor ordered legyen: atrira:@OrderBy(„type”); és lista kell, nem set a sorrend miatt; getsetek átírása; futtatás

másik módszer: legyen külön sorrend nyilvántartó oszlop; int pos atri felvétele a Phonenumberbe; majd az Employeeban a Phonenumber atrira @OrderColumn(name=pos); futtatásra a phoneos táblában megjelent a pos oszlop, 0tól indulva, JPA tartja karban

1. *Egy Employee entitás törlése úgy, hogy törölje a kapcsolt atrikat is*

testRemove(); new Employee, employee.addphone..2x, saveemp(), deleteEmployee(), futtatás; nem működött, mert a törlendő entitásra rekordok hivatkoznak; megoldás:

ne csak a persistálás legyen cascaded, hanem a remove is! @OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REMOVE},

**Kérdések**

*Mit nevezünk kapcsolati attribútumnak?*

*Milyen tulajdonság alapján lehet a kapcsolatokat kategorizálni?*

*Ez alapján milyen fajta kapcsolatok vannak?*

*Mik a gyakrabban használt kombinációk?*

*Hogy állítjuk be ezeket a kapcsolatokat?*

*Mit csinál a getReference() metódus és mire használjuk?*

*Mi az a cascade-olt művelet és milyen fajtái vannak? Hogy állítjuk be?*

*Mi az az orphan removal és mire használjuk? Hogy állítjuk be?*

*Hogy állítunk be sorrendezést lista kapcsolati atriknál? (2 mód)*

*A kapcsolatban hogy hívjuk az oldalakat? Melyik lesz melyik?*

*Mi a lazy kapcsolat?*

*Mi a default betöltési mód a kapcsolatok számosságától függően?*

*Mi az az N+1 probléma, és hogy védjük ki?*

*Mi az a LazyInitializationExcepton, mikor jöhet elő s hogy védjük ki?*

*Hogy lehet megadni, hogy milyen kapcsolódó entitások kerüljenek betöltésre?*

*Hogy deklarálhatjuk ezt?*